Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №7

Технології розробки програмного забезпечення

ШАБЛОН «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»

Виконав:
студент групи ІА-24
Красношапка Р. О.
Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Тема: ШАБЛОН «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»

Короткі теоретичні відомості

Шаблон Mediator (Посередник) є поведінковим шаблоном, який зменшує взаємодії між об'єктами, спрямовуючи всі комунікації через єдиний центральний об'єкт-посередник. Це дозволяє знизити зв'язність між компонентами та спростити їх взаємодію, замінюючи прямі зв'язки між ними на посередницькі. Такий підхід часто використовується в ситуаціях, коли необхідно знизити складність системи з багатьма взаємодіючими елементами.

Facade (Фасад) — це структурний шаблон, який забезпечує спрощений інтерфейс для роботи зі складною підсистемою. Завдяки фасаду клієнт може взаємодіяти з підсистемою через один простий інтерфейс, при цьому внутрішня складність системи приховується. Цей шаблон дозволяє знизити складність використання системи, коли клієнт не потребує доступу до всіх її компонентів.

Шаблон Bridge (Міст) дозволяє розділити абстракцію від її реалізації, що дає можливість змінювати ці дві частини незалежно одна від одної. Це дозволяє створювати гнучкі та масштабовані системи, де нові абстракції та їх реалізації можна додавати без порушення існуючого коду. Це особливо корисно, коли необхідно працювати з різними варіантами реалізації абстракцій, наприклад, для різних платформ чи пристроїв.

Template Method (Шаблонний метод) — це поведінковий шаблон, який визначає загальну структуру алгоритму, дозволяючи підкласам змінювати деякі його частини. В основному класі задається загальна послідовність кроків алгоритму, а підкласи можуть реалізовувати конкретні етапи. Цей шаблон допомагає знизити дублювання коду та забезпечує однакову структуру виконання алгоритмів, що є корисним у фреймворках і бібліотеках.

Хід роботи

Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server) Програмне забезпечення для управління проектами повинно мати наступні функції: супровід завдань/вимог/проектів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

Основні принципи та застосування Bridge (Міст):

Шаблон **Bridge** (Міст) є структурним шаблоном проєктування, який розділяє абстракцію та її реалізацію, дозволяючи змінювати їх незалежно одну від одної. Основний принцип цього шаблону полягає в тому, щоб створити абстракцію, яка містить посилання на реалізацію, а сама реалізація може бути змінена без необхідності змінювати абстракцію. Це дозволяє гнучко адаптувати систему під різні варіанти реалізації.

Шаблон **Bridge** зазвичай використовується в ситуаціях, коли система повинна працювати з кількома варіантами абстракцій та їх реалізацій, і зміна реалізації не повинна впливати на абстракцію, і навпаки. Це допомагає мінімізувати зміну коду в разі зміни однієї з частин, даючи можливість легко додавати нові реалізації або змінювати існуючі.

Застосування шаблону Bridge можна побачити в ситуаціях, де є потреба в змінних або багатофункціональних платформах, що мають різні варіанти реалізації, наприклад, для роботи з різними пристроями або ОС, при цьому зберігаючи однаковий інтерфейс для користувача.

Реалізація шаблону проєктування для майбутньої системи

Шаблон **Bridge** (Міст) є структурним шаблоном проєктування, який дозволяє розділити абстракцію та її реалізацію таким чином, щоб вони могли змінюватися незалежно одна від одної. Цей шаблон створює два окремих рівні: абстракцію, яка визначає високий рівень інтерфейсу, та реалізацію, що відповідає за деталі реалізації цього інтерфейсу. Шаблон Bridge дозволяє змінювати конкретні реалізації без необхідності змінювати абстракцію, що знижує залежність між ними.

```
1   package org.example.projectmanagement.bridge;
2
3   import lombok.AllArgsConstructor;
4   import org.example.projectmanagement.dtos.TaskDto;
5   import org.example.projectmanagement.models.Task;
6
    6 usages  3 inheritors  ± QUIRINO *
7   @AllArgsConstructor
8   public abstract class AbstractTaskBridge implements TaskBridge {
9       protected String methodology;
10      protected TaskBridge taskBridge;
11
12
        no usages  ± QUIRINO *
13      public Task createTask(TaskDto taskDto) {
14          return taskBridge.createTask(taskDto, getInitialStatus(), methodology);
15      }
16
        2 usages  3 implementations  new *
17      @Override
18      public abstract String getInitialStatus();
19  }
```

Рис. 1 – Код інтерфейсу AbstractTaskBridge

```java
package org.example.projectmanagement.bridge.impl;

import org.example.projectmanagement.bridge.AbstractTaskBridge;
import org.example.projectmanagement.bridge.TaskBridge;
import org.example.projectmanagement.dtos.TaskDto;
import org.example.projectmanagement.models.Task;
import org.example.projectmanagement.models.enums.RUPPhase;

2 usages  QUIRINO *
public class RupTaskBridge extends AbstractTaskBridge {

    1 usage  new *
    public RupTaskBridge(TaskBridge taskBridge) {
        super( methodology: "RUP", taskBridge);
    }

    2 usages  QUIRINO *
    @Override
    public String getInitialStatus() {
        return RUPPhase.INCEPTION.name();
    }

    5 usages  new *
    @Override
    public Task createTask(TaskDto taskDto, String initialStatus, String methodology) {
        return taskBridge.createTask(taskDto, initialStatus, methodology);
    }
}
```

Рис. 2 – Код класу RupTaskBridge

```java
package org.example.projectmanagement.bridge.impl;

import org.example.projectmanagement.bridge.AbstractTaskBridge;
import org.example.projectmanagement.bridge.TaskBridge;
import org.example.projectmanagement.dtos.TaskDto;
import org.example.projectmanagement.models.Task;
import org.example.projectmanagement.models.enums.KanbanColumn;

2 usages  QUIRINO *
public class KanbanTaskBridge extends AbstractTaskBridge {

    1 usage  new *
    public KanbanTaskBridge(TaskBridge taskBridge) {
        super( methodology: "Kanban", taskBridge);
    }

    2 usages  QUIRINO *
    @Override
    public String getInitialStatus() {
        return KanbanColumn.BACKLOG.name();
    }

    5 usages  new *
    @Override
    public Task createTask(TaskDto taskDto, String initialStatus, String methodology) {
        return taskBridge.createTask(taskDto, initialStatus, methodology);
    }
}
```

Рис. 3 – Код класу KanbanTaskBridge

```java
package org.example.projectmanagement.bridge.impl;

import org.example.projectmanagement.bridge.AbstractTaskBridge;
import org.example.projectmanagement.bridge.TaskBridge;
import org.example.projectmanagement.dtos.TaskDto;
import org.example.projectmanagement.models.Task;
import org.example.projectmanagement.models.enums.AgileStatus;

2 usages  ▲ QUIRINO *
public class AgileTaskBridge extends AbstractTaskBridge {

    1 usage  new *
    public AgileTaskBridge(TaskBridge taskBridge) {
        super( methodology: "Agile", taskBridge);
    }

    2 usages  ▲ QUIRINO *
    @Override
    public String getInitialStatus() {
        return AgileStatus.NEW.name();
    }

    5 usages  new *
    @Override
    public Task createTask(TaskDto taskDto, String initialStatus, String methodology) {
        return taskBridge.createTask(taskDto, initialStatus, methodology);
    }
}
```

Рис. 4 – Код класу AgileTaskBridge

```java
package org.example.projectmanagement.bridge;

import org.example.projectmanagement.dtos.TaskDto;
import org.example.projectmanagement.models.Task;

12 usages  4 implementations  ▲ QUIRINO *
public interface TaskBridge {
    5 usages  3 implementations  new *
    Task createTask(TaskDto taskDto, String initialStatus, String methodology);
    2 usages  4 implementations  new *
    String getInitialStatus();
}
```

Рис. 5 – Код interface TaskBridge

```java
@Override
public TaskDto addTaskToProject(String projectId, TaskDto taskDto, List<MultipartFile> files) throws IOException {
    projectValidator.validateId(projectId);
    Project project = projectRepository.findById(projectId)
            .orElseThrow(() -> new IllegalArgumentException("Project not found: " + projectId));

    TaskBridge taskBridge = TaskBridgeProvider.getBridge(project.getMethodology(), taskBridge: null);
    Task task = taskBridge.createTask(taskDto, taskBridge.getInitialStatus(), project.getMethodology());
    Task savedTask = taskService.saveTask(task, files);

    project.addTaskId(savedTask.getTaskId());
    projectRepository.save(project);
    return taskConverter.buildTaskDtoFromTask(savedTask);
}
```

Рис. 6 – Приклад використання TaskBridge

У коді паттерн **"Міст" (Bridge)** використовується для створення задач (Task) в різних методологіях, таких як Agile, Kanban і RUP. Цей підхід дозволяєрозділити абстракцію від її реалізації, що забезпечує гнучкість у зміні обох без необхідності переписувати основний код.

Принцип роботи паттерна полягає в тому, що **TaskBridge** визначає інтерфейс з методами, які мають бути реалізовані в конкретних реалізаціях моста. Ці методи, такі як **createTask** та **getInitialStatus**, відповідають за створення задач і визначення їх початкового статусу. Абстрактний клас **AbstractTaskBridge** реалізує цей інтерфейс, надаючи базову логіку, спільну для всіх методологій, і може бути розширений для впровадження специфічної поведінки в підкласах.

Конкретні реалізації мостів, такі як **AgileTaskBridge**, **KanbanTaskBridge** і **RupTaskBridge**, розширюють **AbstractTaskBridge** та додають специфічну логіку, адаптуючи методи для кожної методології. Наприклад, **AgileTaskBridge** може реалізувати метод **createTask** з урахуванням принципів Agile, тоді як **KanbanTaskBridge** — для Kanban, і так далі для RUP.

Клас **TaskBridgeProvider** виступає провайдером для мостів, відповідаючи за надання конкретної реалізації моста залежно від вибраної методології. Використовуючи метод **getBridge**, цей клас забезпечує динамічний вибір реалізації, не змінюючи основний код програми.

У методі **addTaskToProject** клас **TaskBridgeProvider** отримує відповідний міст для вибраної методології і за допомогою цього моста створюється задача з урахуванням специфічної логіки для кожної методології. Це дозволяє зберегти гнучкість у системі, де методології можна змінювати без необхідності переписувати значну частину коду, що забезпечує масштабованість і зручність підтримки програми.

Зображення структури шаблону

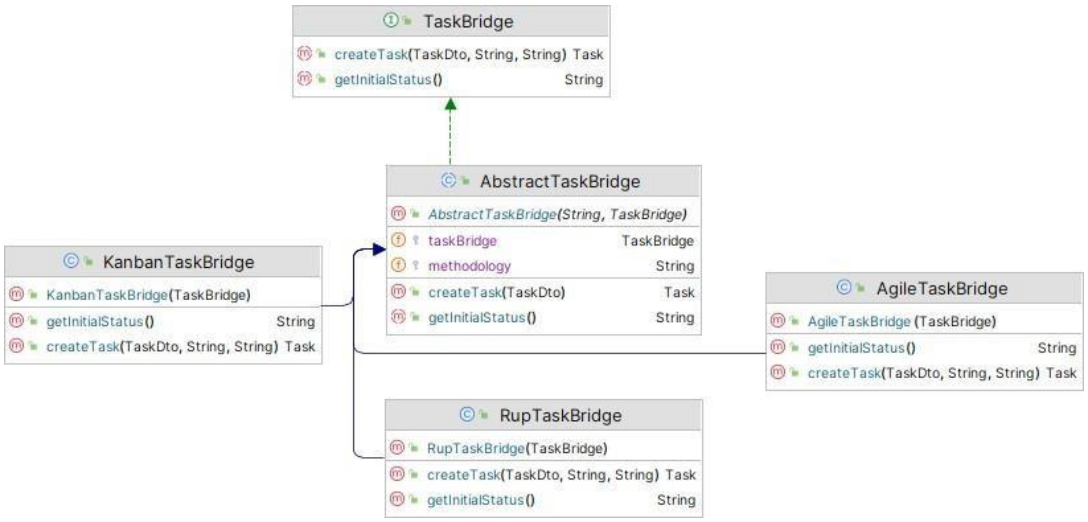

Рис. 7 – Структура шаблону

Посилання на репозиторій:

https://github.com/QUIRINO228/projectManagmentSoftware

Висновок: У результаті реалізації шаблону Bridge було продемонстровано ефективність цього патерна для розділення абстракції та її реалізації, що дозволяє змінювати обидві незалежно одна від одної. Це забезпечує гнучкість у виборі необхідних реалізацій в залежності від контексту, дозволяючи легко

додавати нові платформи або компоненти без необхідності змінювати логіку клієнта.

За допомогою шаблону Bridge абстракція та її конкретні реалізації взаємодіють через спільний інтерфейс, що дозволяє створювати задачі, компоненти або функціональність для різних методологій чи платформ без тісної прив'язки до конкретних реалізацій. Цей підхід значно спрощує розширення системи, оскільки додавання нових варіантів продуктів або платформ не вимагає переписування основної логіки, зберігаючи систему чистою і масштабованою.